# Git Merge VS Git Rebase

**What is Git Merge?**

Git merge integrates two branches into one, combining the commit histories of both branches. It creates a new commit (called a **merge commit**) if the branches diverged, preserving the complete commit history.

**Types of Merge**

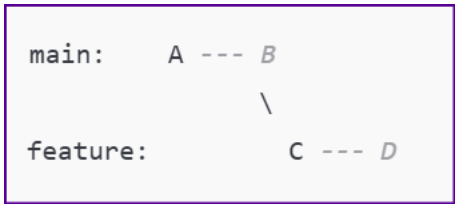
**1. Fast-Forward Merge**

A fast-forward merge happens when the target branch has no new commits since the source branch diverged. The merge simply moves the branch pointer forward.

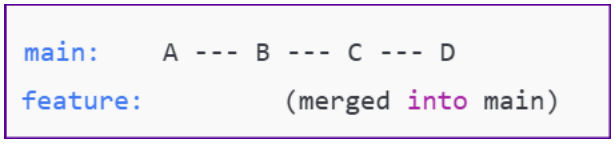
---

**Scenario 1: Fast-Forward Merge**

**Branch State Before Merge:**




**Fast-Forward Merge Result:**




**Commands:**

```
# Initialize a Git repository
git init

# Create an initial commit on main
echo "Initial commit" > file1
git add file1
git commit -m "Initial Commit A"
git checkout -b feature  # Create and switch to 'feature'

# Add commits to feature branch
echo "Feature work 1" > file2
git add file2
git commit -m "Commit C"
echo "Feature work 2" > file3
git add file3
git commit -m "Commit D"

# Switch back to main and perform a fast-forward merge
```

```
git checkout main
git merge feature  # Fast-forward merge
```

---

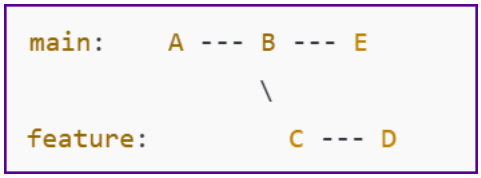
**2. Three-Way Merge**

A three-way merge occurs when both branches have made commits since they diverged. Git creates a **merge commit** to integrate the histories.

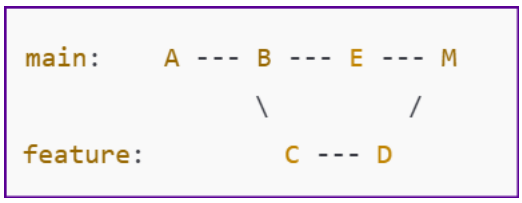
---

**Scenario 2: Three-Way Merge**

**Branch State Before Merge:**




**Three-Way Merge Result:**




**Commands:**

```
# Create a new repository
git init

# Create an initial commit on main
echo "Initial commit" > file1
git add file1
git commit -m "Commit A"

# Create a feature branch and add commits
git checkout -b feature
echo "Feature work 1" > file2
git add file2
git commit -m "Commit C"
echo "Feature work 2" > file3
git add file3
git commit -m "Commit D"

# Switch back to main and add a commit
git checkout main
echo "Main branch work" > file4
git add file4
git commit -m "Commit E"

# Perform a three-way merge
```

git merge feature

---

**Advantages of Merge**

1. **Preserves History**: Retains all commits in both branches.
2. **Conflict Management**: Easier to resolve conflicts at merge time.
3. **Context**: Provides clear context for branch integration with merge commits.

---

**Drawbacks of Merge**

1. **Non-Linear History**: Creates a graph-like history with multiple branches.
2. **Potential Clutter**: Repeated merges can lead to a cluttered commit graph.

---

**Git Rebase**

**What is Git Rebase?**

Git rebase rewrites the commit history by **replaying commits** from the current branch onto the target branch. This process eliminates merge commits, creating a clean, linear history.

---

**Scenario 3: Rebase to Update a Branch**

**Branch State Before Rebase:**

```
main:    A --- B --- E
                \
feature:          C --- D
```

**Rebase Result:**

```
main:    A --- B --- E --- C' --- D'
```

---

**How Rebase Works**

1. Git identifies the **common ancestor** between main and feature (in this case, B).
2. It temporarily stores the commits from feature (C and D) as patches.
3. It applies the commits from main (E) onto feature.
4. Finally, it reapplies the stored commits (C and D) on top of E.

---

**Commands:**

```
# Start with a repository having two branches
git init
echo "Initial commit" > file1
git add file1
git commit -m "Commit A"

# Add commits on main
echo "Main branch work 1" > file2
git add file2
git commit -m "Commit B"

# Create a feature branch and add commits
git checkout -b feature
echo "Feature work 1" > file3
git add file3
git commit -m "Commit C"
echo "Feature work 2" > file4
git add file4
git commit -m "Commit D"

# Switch back to main and add another commit
git checkout main
echo "Main branch work 2" > file5
git add file5
git commit -m "Commit E"

# Rebase feature onto main
git checkout feature
git rebase main
```

---

**Rebase with Conflict**

When rebasing, conflicts may occur if the same lines in a file were modified in both branches. Git pauses the rebase process to let you resolve the conflict.

**Steps to Resolve Conflicts:**

1. Git will stop and indicate the conflicted files.
2. Open the conflicted files and manually resolve the conflicts.
3. Add the resolved files back to the staging area using git add.
4. Continue the rebase process using git rebase --continue.

---

**Advantages of Rebase**

1. **Clean History**: Results in a linear, easy-to-read commit history.
2. **Compact Commits**: Useful for small teams or personal projects.

3. **Avoids Merge Commits**: Simplifies the commit graph.

---

**Drawbacks of Rebase**

1. **History Rewriting**: Dangerous if applied to public/shared branches.
2. **Conflict Resolution**: Can be tedious for large commits or branches.
3. **Potential Data Loss**: Improper use can accidentally discard commits.

---

**Merge vs Rebase: Detailed Comparison**

| **Feature** | **Git Merge** | **Git Rebase** |
|---|---|---|
| **History Structure** | Retains a branch graph with merge commits. | Creates a linear, rewritten history. |
| **Use Case** | Preserves original context for collaboration. | Simplifies history for personal or small-team projects. |
| **Commit Timestamps** | Maintains original timestamps. | Updates commit timestamps (new commits). |
| **Conflict Handling** | Conflicts resolved during merge. | Conflicts resolved during rebase process. |
| **Public/Shared Branch** | Safe for shared branches. | Unsafe for shared branches (rewrites history). |
| **Resulting History** | Non-linear (merge commits are visible). | Linear (no merge commits). |

---

**When to Use Merge or Rebase**

1. **Use Merge**:
    - Collaborative projects where history context is important.
    - Large teams where merge commits clarify integration points.
    - When avoiding history rewriting is critical.
2. **Use Rebase**:
    - For cleaning up a messy commit history in personal projects.
    - Before merging your feature branch into the main branch.
    - To keep the project history linear and easy to understand.